REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

AD-A236 412

1. AG

DATE

3. REPORT TYPE AND DATES COVERED
Final: 15 Aug 1990 to 01 Mar 1993

4. TITLE AND SUBTITLE
Alsys Limited, AlsyCOMP_023, Version 5.3, IBM 370 3084Q (under MVS/XA release 3.2)(Host & Target), 901125N1.11072

5. FUNDING NUMBERS

6. AUTHOR(S)
National Computing Centre Limited
Manchester, UNITED KINGDOM

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Computing Centre Limited
Oxford Road
Manchester MI 7ED
UNITED KINGDOM

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF_VSR_90502/77-910404

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Alsys Limited, AlsyCOMP_023, Version 5.3, Manchester England, IBM 370 3084Q (under MVS/XA release 3.2), ACVC 1.11.

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on **901125.**

Compiler Name and Version:	**AlsyCOMP_023 Version 5.3**
Host Computer System:	**IBM 370 3084Q (under MVS/XA release 3.2)**
Target Computer System:	**IBM 370 3084Q (under MVS/XA release 3.2)**

A more detailed description of this Ada implementation is found in section 3.1 of this report. As a result of this validation effort, Validation Certificate **#901125N1.11072** is awarded to **Alsys Limited.** This certificate expires on **01 JUNE 1992.**

This report has been reviewed and is approved.

A.E.J. Pink

Jane Pink
Testing Services Manager
The National Computing Centre Limited
Oxford Road
Manchester
M1 7ED
England

for

Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria
VA 22311

for William Curtis Collett

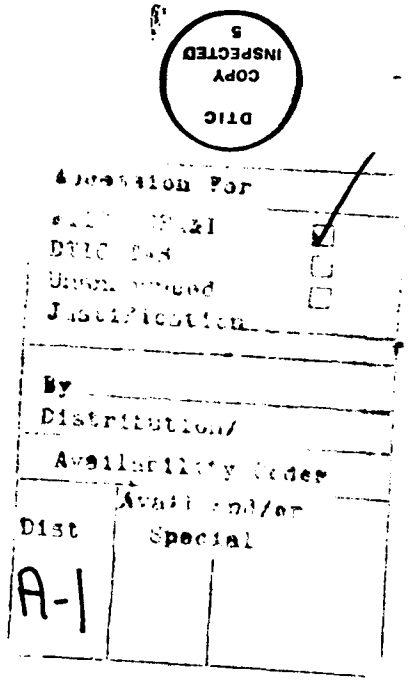
Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington
DC 20301

91-00491

AVF Control Number: AVF_VSR_90502/77-910404

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #901125N1.11072
Alsys Limited
AlsyCOMP_023 Version 5.3
IBM 370 3084Q (under MVS/XA release 3.2)

Prepared by
Testing Services
The National Computing Centre Limited
Oxford Road
Manchester
M1 7ED
England



VSR Version 90-08-15

The following declaration of conformance was supplied by the customer.

DECLARATION OF CONFORMANCE

Customer: **Alsys Limited**

Ada Validation Facility: **The National Computing Centre Limited**
Oxford Road
Manchester
M1 7ED
United Kingdom

ACVC Version: **1.11**

Ada Implementation:

Ada Compiler Name: **AlsyCOMP_023**

Version: **Version 5.3**

Host Computer System: **IBM 370 3084Q (under MVS/XA release 3.2)**

Target Computer System: **IBM 370 3084Q (under MVS/XA release 3.2)**

Customer's Declaration

I, the undersigned, representing **Alsys Limited**, declare that **Alsys Limited** has no knowledge of deliberate deviations from the Ada Language Standard **ANSI/MIL-STD-1815A** in the implementation(s) listed in this declaration.

Signature

23 - 11 - 90

Date

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield
VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria
VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987

[Pro90] Ada Compiler Validation Procedures,
Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide,
21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behaviour is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3. For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada Certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several interconnected units.
Conformity	Fulfilment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.

Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 90-10-12.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026B
B85001L	C83026A	C83041A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Issues and commonly referenced in the format AI-dddd. For this implementation, the following tests were inapplicable for the reasons indicated; references to Ada Issues are included as appropriate.

The following 159 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113O..Y (11 tests)	C35705O..Y (11 tests)
C35706O..Y (11 tests)	C35707O..Y (11 tests)
C35708O..Y (11 tests)	C35802O..Z (12 tests)
C45241O..Y (11 tests)	C45321O..Y (11 tests)
C45421O..Y (11 tests)	C45521O..Z (12 tests)
C45524O..Z (12 tests)	C45621O..Z (12 tests)
C45641O..Y (11 tests)	C46012O..Z (12 tests)

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45423A checks that the proper exception is raised if MACHINE_OVERFLOWS is TRUE for the floating point type FLOAT.

C45423B checks that the proper exception is raised if MACHINE_OVERFLOWS is TRUE for the floating point type SHORT-FLOAT

C45523A and C45622A check that the proper exception is raised if MACHINE_OVERFLOWS is TRUE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS if FALSE.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45536A, C46013B, C46031B, C46033B and C46034B contain 'SMALL representation clauses which are not powers of two or ten.

C86001F recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete. For this implementation, the package TEXT_IO is dependent upon package SYSTEM.

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A53A checks operations of a fixed-point type for which a length clause specified a power-of-ten type'small; this implementation does not support decimal smalls. (See 2.3).

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2107B..E (4 tests), CE2107L, CE2110B and CE2111D attempt to associate multiple internal files with the same external file when one or more files is writing, or reading and writing for sequential files. The proper exception is raised when multiple access is attempted.

CE2107G..H (2 tests), CE2110D, and CE2111H attempt to associate multiple internal files with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

EE2401D checks that instantiations for DIRECT_IO for unconstrained types are supported. This implementation requires a FORM parameter to be used to specify the maximum runtime size of any value of the type for which IO is to be performed.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A attempt to associate multiple internal files with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for **26** tests.

C64103A and C95084A were graded passed by Evaluation Modification as directed by the AVO. Because this implementation's actual values for LONG_FLOAT'SAFE_LARGE and SHORT_FLOAT'LAST lie within one (SHORT_FLOAT) model interval of each other, the tests' floating-point applicability check may evaluate to TRUE and yet the subsequent expected exception need not be raised. The AVO ruled that the implementation's behaviour should be graded as passed because the implementation passed the integer and fixed-point checks; the following REPORT.FAILED messages were produced after the type conversions at line 198 in C64103A and lines 101 and 250 in C95084A failed to raise exceptions:

C64103A: "EXCEPTION NOT RAISED AFTER CALL -P2 (B)"

C95084A: "EXCEPTION NOT RAISED BEFORE CALL - T2 (A)"
"EXCEPTION NOT RAISED AFTER CALL - T5 (B)"

EA3004D was graded passed by Evaluation and Processing Modification as directed by the AVO. The test requires that either pragma INLINE is obeyed for a function call in each of three contexts and that thus three library units are made obsolete by the re-compilation of the inlined function's body, or else the pragma is ignored completely. This implementation obeys the pragma except when the call is within a package specification. When the test's files are processed in the given order, only two units are made obsolete; thus, the expected error at line 27 of file EA3004D6M is not valid and is not flagged. To confirm that indeed the pragma is not obeyed in this one case, the test was also processed with the files re-ordered so that the re-compilation follows only the package declaration (and thus the other library units will not be made obsolete, as they are compiled later); a "NOT APPLICABLE" result was produced, as expected. The revised order of files was 0-1-4-5-2-3-6.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-10 value as small for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B23004A B24007A B24009A B28003A

B32202A	B32202B	B32202C	B37004A
B45102A	B61012A	B74304A	B74401F
B74401R	B91004A	B95069A	B95069B
B97103E	BA1101B2	BA1101B4	BC2001D
BC3009C	BC3204D		

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Jon Frosdick
Alsys Limited
Partridge House
Newtown Road
Henley-on-Thames
Oxfordshire
RG9 1EN

For a point of contact for sales information about this Ada implementation system, see:

John Stewart
Alsys Limited
Partridge House
Newtown Road
Henley-on-Thames
Oxfordshire
RG9 1EN

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a)	Total Number of Applicable Tests	3834
b)	Total Number of Withdrawn Tests	81
c)	Processed Inapplicable Tests	255
d)	Non-Processed I/O Tests	0

e)	Non-Processed Floating-Point Precision Tests	0	
f)	Total Number of Inapplicable Tests	255	(c+d+e)
g)	Total Number of Tests for ACVC 1.11	4170	(a+b+f)

All I/O tests of the test suite were processed because this implementation supports a file system. All floating-point precision tests were processed because this implementation supports floating-point precision to the extent that was tested. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

A **Magnetic Tape** containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the **Magnetic Tape** were loaded onto a SUN 3/160 and then transferred to an IBM 9370 Model 90 running under VM/IS CMS. A tape readable by the host was then created on the IBM 9370.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

CALLS=INLINED	Allows inline insertion of subprogram code
REDUCTION=EXTENSIVE	Perform extensive high level optimisations
EXPRESSIONS=EXTENSIVE	Perform extensive low level optimisations
OBJECT=PEEPHOLE	Perform peephole optimisations

In addition the following options were used to produce full compilation listings including source text.

TEXT	Include full source text in listing
WARNING=NO	Do not include warning messages in listing
DETAIL=NO	Do not add extra detail to error messages
SHOW=NONE	Do not print page headers or error summaries
ERROR=999	Stop after 999 errors
FILE_WIDTH=79	Set width of listing file to 79 columns

FILE_LENGTH=9999	Disable insertion of form feeds in listing
OUTPUT=<file>	Send listing to specified file name

Test output, compiler and linker listings, and job logs were captured on Magnetic Tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the valued that are defined in terms of the maximum input-line length, which is the value for $MAX_IN-LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	255
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	4294967296
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	5370
$DELTA_DOC	2:1.0:E-31
$ENTRY_ADDRESS	SYSTEM.NULL_ADDRESS
$ENTRY_ADDRESS1	SYSTEM.NULL_ADDRESS
$ENTRY_ADDRESS2	SYSTEM.NULL_ADDRESS
$FIELD_LAST	255
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	"LRECL=>80,RECFM=F"
$FORM_STRING2	CANNOT_RESTRICT_FILE_CAPACITY
$GREATER_THAN_DURATION	100000.0
$GREATER_THAN_DURATION_BASE_LAST	10000000.0
$GREATER_THAN_FLOAT_BASE_LAST	1.0E+80
$GREATER_THAN_FLOAT_SAFE_LARGE	16:0.FFFF_FFFF_FFFF_F9:E63

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	16:0.FFFF_F9:E63
$HIGH_PRIORITY	10
$ILLEGAL_EXTERNAL_FILE_NAME1	T??????? LISTING A1
$ILLEGAL_EXTERNAL_FILE_NAME2	TOOLONGNAME TOOLONGTYPE TOOLONGMODE
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A23006D1 TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006D1 TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	ASSEMBLER
$LESS_THAN_DURATION	-100000.0
$LESS_THAN_DURATION_BASE_FIRST	-10000000.0
$LINE_TERMINATOR	' '
$LOW_PRIORITY	1
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NO_SUCH_TYPE
$MANTISSA_DOC	31
$MAX_DIGITS	18
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648
$MIN_INT	-2147483648

$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	I80X86,I80386,MC680X0,S370,TRANSPUTER,VAX
$NAME_SPECIFICATION1	'TERALG.X2120A'
$NAME_SPECIFICATION2	'TERALG.X2120B'
$NAME_SPECIFICATION3	'TERALG.X3119A'
$NEG_BASED_INT	16:FFFFFFFF:
$NEW_MEM_SIZE	0
$NEW_STOR_UNIT	0
$NEW_SYS_NAME	I80X86 I80386 MC680X0 TRANSPUTER VAX
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	NEW INTEGER;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	10240
$TICK	0.01
$VARIABLE_ADDRESS	V_ADDRESS
$VARIABLE_ADDRESS1	V_ADDRESS1
$VARIABLE_ADDRESS2	V_ADDRESS2
$YOUR_PRAGMA	RMODE

APPENDIX B

COMPILATION SYSTEM OPTIONS

Include a separate list of options and their meanings for each of the software systems used in this validation. A software system must be the compiler and could be the linker, the loader, the binder, etc. (Version numbers should be included)

Compiler Options

SOURCE=*file_name*	The name of the source file.
LIBRARY=*library_name*	The name of the Ada program library.
ANNOTATE=""	User specified character string annotating compilation unit as stored in library.
LEVEL=UPDATE	Compilation level - complete compilation of source code into object code and update of program library.
ERRORS=999	Number of errors permitted before compilation is terminated.
CHECKS=ALL	All run time checks to be performed, except those explicitly suppressed by use of pragma SUPPRESS.
GENERICS=INLINE	Place code of generics instantiations inline in the same unit as the instantiation rather than in separate units.
MEMORY=500	Number of Kbytes reserved in memory for compiler data (before swapping commences).
OUTPUT=*file_name*	Compilation listing file name.
TEXT=YES or NO	Controls inclusion of full source test in the compilation listing. Set to YES for tests requiring compilation listings (ie B tests). Set to NO for tests not requiring compilation listings (ie non-B tests).
WARNING=NO	Do not include warning messages in the compilation listing.
SHOW=NONE	Do not print a header on compilation listing pages, nor an error summary at the end.
DETAIL=NO	Do not print extra detail in error messages in the compilation listing.

ASSEMBLY=NONE	Do not include an assembly listing of generated code in the compilation listing.
STACK=1024	Maximum size in bytes for objects allocated in the static part of a stack frame. Objects bigger than this limit are allocated in the dynamic part of a stack frame.
GLOBAL=1024	Maximum size in bytes for objects allocated in the global data area of a compilation unit. Objects bigger than this limit are allocated on the program heap.
UNNESTED=16	Maximum size in bytes for objects allocated in the stack frame of the enclosing unit of a separately compiled package body. Objects bigger than this limit are allocated in the frame of the separate package body itself.
CALLS=INLINED	Allow inline insertion of code for subprograms.
REDUCTION=EXTENSIVE or NONE	Controls the optimisation of run-time checks and remove dead code. Set to EXTENSIVE for AlsyCOMP_006, implying full optimisation. Set to NONE for AlsyCOMP_023, implying no optimisation.
EXPRESSIONS=EXTENSIVE or NONE	Controls the optimisation of expression evaluation. Set to extensive for AlsyCOMP_006, implying full optimisation. Set to NONE for AlsyCOMP_023, implying no optimisation.
OBJECT=PEEPHOLE	Optimise locally the object code as it is generated.
COPY=NO	Do not save a representation of the source code in the program library.
DEBUG-NO	Do not save information for debugging.
TREE=NO	Do not save information for cross referencing.
FILE_WIDTH=79	Width of compilation listing page in columns.
FILE_LENGTH=9999	Length of compilation listing page in lines (effectively unpaginated).
Binder Options	
PROGRAM=*unit_name*	The name of the main unit of the Ada program.
LIBRARY=*library_name*	The name of the Ada program library.

LEVEL=BIND	Binding level - complete bind to produce an object module.
OBJECT=*file name*	Name of generated object module.
UNCALLED=REMOVE	Remove the code for uncalled subprograms from the load module.
SLICE=1000	Preform timeslicing, invoking the task scheduler every 1999 milliseconds.
MAP_TASKS=NONE	Do not explicitly map tasks to operating system processes.
MAIN=64	Number of Kbytes initially allocated to the main program stack.
TASK=16	Default number of Kbytes initial allocated to task stacks (in absence of explicit length clause).
HISTORY=NO	Do not provide a full trace of the propagation of exceptions unhandled in the main program.
SIZE=256	Number of Kbytes initially allocated to the program heap.
INCREMENT=4	Quantum size, expressed in Kbytes, by which the size of the program heap is incremented upon exhaustion.
OUTPUT=*file_name*	Binder listing file name.
DATA=NO	Do not print additional mapping information in the binder listing.
WARNING=NO	Do not print warning messages in the binder listing.
DEBUG=NO	Do not save information for debugging.
CUI_FILE=AUTOMATIC	Name of file in which debugging information would be stored (if generated) would be derived automatically from PROGRAM name.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

MODE RMODE (ANY)

link program to run in XA or non XA mode

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

type INTEGER is range -2147483648 .. 2147483647;
type SHORT_INTEGER is range -32768 .. 32767;
type SHORT_SHORT_INTEGER is range -128 .. 127;

type FLOAT is digits 15 range -7.24E+75 .. 7.24E+75;
type SHORT_FLOAT is digits 6 range -7.24E+75 .. 7.24E+75;
type LONG_FLOAT is digits 18 range -7.24E+75 .. 7.24E+75;

type DURATION is delta 2.0**-14 range -131072.0 .. 131071.0

end STANDARD;
```

Alsys IBM 370 Ada Compiler

APPENDIX F

for VM/CMS, MVS and MVS/XA

Implementation - Dependent Characteristics

Version 5

Alsys S.A.
29, Avenue Lucien-René Duschesne
78170 La Celle St. Cloud, France

Alsys GmbH
Am Rüppurrer Schloß 7
D-7500 Karlsruhe 51, Germany

Alsys Inc.
67 South Bedford Street
Burlington, MA 01803-5152, U.S.A.

Alsys AB
Patron Pehr Väg 10
Box 1085
141 22 Huddinge, Stockholm, Sweden

Alsys Ltd
Partridge House, Newtown Road
Henley-on-Thames
Oxon, RG9 1EN, U.K.

Alsys KKE Co., Ltd
TechnoWave 100, 16F
1-1-25 Shin-Urashima-cho
Kanagawa-ku
Yokohama #221, Japan

Printed: October 1990

PREFACE

This *Alsys IBM 370 Ada Compiler Appendix F for* VM/CMS, MVS and MVS/XA is for programmers, software engineers, project managers, educators and students who want to develop an Ada program for any IBM System/370 processor that runs VM/CMS, MVS or MVS/XA.

This appendix is a required part of the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815A, January 1983 (throughout this appendix, citations in square brackets refer to this manual). It assumes that the user is already familiar with the VM/CMS, MVS and MVS/XA operating system, and has access to the following IBM documents:

CMS User Guide, SC19-6210

CMS Command and Macro Reference, SC19-6209

OS/VS2 MVS Overview, GC28-0984

OS/VS2 System Programming Library: Job Management, GC28-1303

MVS/370 JCL Reference, GC28-1350

IBM System/370 Principles of Operation, GA22-7000

IBM System/370 System Summary, GA22-7001

TABLE OF CONTENTS

INTRODUCTION

Implementation-Dependent Characteristics

This appendix summarizes the implementation-dependent characteristics of the Alsys IBM 370 Ada Compiler for VM/CMS, MVS and MVS/XA. This document should be considered as the Appendix F to the Reference Manual for the Ada Programming Language ANSI/MIL-STD 1815A, January 1983, as appropriate to the Alsys Ada implementation for the IBM 370 under VM/CMS, MVS and MVS/XA.

Sections 1 to 8 of this appendix correspond to the various items of information required in Appendix F [F]*; sections 9 and 10 provide other information relevant to the Alsys implementation. The contents of these sections is described below:

1. The form, allowed places, and effect of every implementation-dependent pragma.

2. The name and type of every implementation-dependent attribute.

3. The specification of the package SYSTEM [13.7].

4. The list of all restrictions on representation clauses [13.1].

5. The conventions used for any implementation-generated names denoting implementation-dependent components [13.4].

6. The interpretation of expressions that appear in address clauses, including those for interrupts [13.5].

7. Any restrictions on unchecked conversions [13.10.2].

8. Any implementation-dependent characteristics of the input-output packages [14].

9. Characteristics of numeric types.

---

* Throughout this manual, citations in square brackets refer to the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, January 1983.

10. Other implementation-dependent characteristics.

Throughout this appendix, the name *Ada Run-Time Executive* refers to the run-time library routines provided for all Ada programs. These routines implement the Ada heap, exceptions, tasking control, I/O, and other utility functions.

CHAPTER 1

Implementation-Dependent Pragmas

1.1 INLINE

Pragma INLINE is fully supported, except for the fact that it is not possible to inline a function call in a declarative part. Control of inlining is also possible using the COMPILE command with the option IMPROVE (see the *User's Guide*, Chapter 4).

1.2 INTERFACE

Ada programs can interface to subprograms written in another language through the use of the predefined pragma INTERFACE [13.9] and the implementation-defined pragma INTERFACE_NAME.

Pragma INTERFACE specifies the name of an interfaced subprogram and the name of the programming language for which calling and parameter passing conventions will be generated. Pragma INTERFACE takes the form specified in the *Reference Manual*:

pragma INTERFACE (*language_name*, *subprogram_name*);

where:

- *language_name* is the name of the other language whose calling and parameter passing conventions are to be used.
- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.

The only language name currently accepted by pragma INTERFACE is ASSEMBLER.

The language name used in the pragma INTERFACE does not necessarily correspond to the language used to write the interfaced subprogram. It is used only to tell the Compiler how to generate subprogram calls, that is, which calling conventions and parameter passing techniques to use.

The language name ASSEMBLER is used to refer to the standard IBM 370 calling and parameter passing conventions. The programmer can use the language name ASSEMBLER to interface Ada subprograms with subroutines written in any language that follows the standard IBM 370 calling conventions.

1.3 INTERFACE_NAME

Pragma INTERFACE_NAME associates the name of an interfaced subprogram, as declared in Ada, with its name in the language of origin. If pragma INTERFACE_NAME is not used, then the two names are assumed to be identical.

This pragma takes the form:

pragma INTERFACE_NAME (*subprogram_name*, *string_literal*);

where:

- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.
- *string_literal* is the name by which the interfaced subprogram is referred to at link-time.

The use of INTERFACE_NAME is optional, and is not needed if a subprogram has the same name in Ada as in the language of origin. It is necessary, for example, if the name of the subprogram in its original language contains characters that are not permitted in Ada identifiers. Ada identifiers can contain only letters, digits and underscores, whereas the IBM 370 linkage editor/loader allows external names to contain other characters, e.g. the plus or minus sign. These characters can be specified in the *string_literal* argument of the pragma INTERFACE_NAME.

The pragma INTERFACE_NAME is allowed at the same places of an Ada program as the pragma INTERFACE [13.9]. However, the pragma INTERFACE_NAME must always occur after the pragma INTERFACE declaration for the interfaced subprogram.

In order to conform to the naming conventions of the IBM 370 linkage editor/loader, the link-time name of an interfaced subprogram will be truncated to 8 characters and converted to upper case.

Example

```
package SAMPLE_DATA is
    function SAMPLE_DEVICE  (X : INTEGER) return INTEGER;
    function PROCESS_SAMPLE (X : INTEGER) return INTEGER;
    private
        pragma INTERFACE (C, SAMPLE_DEVICE);
        pragma INTERFACE (C, PROCESS_SAMPLE);
        pragma INTERFACE_NAME (PROCESS_SAMPLE, "PSAMPLE");
end SAMPLE_DATA;
```

1.4 EXPORT

The pragma export takes a language name and an Ada identifier as arguments. This pragma allows an object defined in Ada to be visible to external programs written in the specified language.

pragma EXPORT(*language_name, Ada_identifier*)

Example:

```
package MY_PACKAGE is

    THIS_OBJECT:INTEGER;
    pragma EXPORT (ASSEMBLER, THIS_OBJECT);
    ......

end MY_PACKAGE;
```

The language names supported are the same as those supported by pragma INTERFACE.

Limitations on the use of pragma EXPORT

- This pragma must occur in a declarative part and applies only to objects declared in this same declarative part, that is, generic instantiated objects or renamed objects are excluded.
- The pragma is only to be used for objects with direct allocation mode, which are declared in a library package. The ALSYS implementation gives indirect allocation mode to dynamic objects and objects that have significant size (see *Application Developer's Guide,* Chapter 2.

1.5 EXTERNAL_NAME

To name an exported Ada object as it is identified in the other language may require the use of non-Ada naming conventions, such as special characters, or case sensitivity. For this purpose the implementation-dependent pragma EXTERNAL_NAME may be used in conjunction with the pragma EXPORT:

pragma EXTERNAL_NAME (*Ada_identifier, name_string*);

The *name_string* is a string which denotes the name of the identifier defined in the other language. The *Ada_identifier* denotes the exported Ada object.

The pragma EXTERNAL_NAME may be used anywhere in an Ada program where pragma EXPORT is allowed. It must occur after the corresponding pragma EXPORT and within the same library package.

Example:

```
package MY_PACKAGE is

    THIS_OBJECT:INTEGER;
    pragma EXPORT (ASSEMBLER, THIS_OBJECT);
    pragma EXTERNAL_NAME (THIS_OBJECT, "THISOBJ");
    .....

end MY_PACKAGE;
```

1.6 INDENT

This pragma is only used with the Alsys Reformatter (*AdaReformat*); this tool offers the functionalities of a source reformatter in an Ada environment.

The pragma is placed in the source file and interpreted by the Reformatter.

pragma INDENT(OFF)

The Reformatter does not modify the source lines after the OFF pragma INDENT.

pragma INDENT(ON)

The Reformatter resumes its action after the ON pragma INDENT. Therefore any source lines that are bracketed by the OFF and ON pragma INDENTs are not modified by the Alsys Reformatter.

1.7 RMODE

Pragma RMODE associates a residence mode with the objects designated by the access values belonging to a given access type.

This pragma takes the form:

pragma RMODE (*access_type_name*, *residence_mode*);

residence_mode ::= **A24** | **ANY**

where:

- *access_type_name* is the name of the access type defining the collection of objects whose residence mode is to be specified.
- *residence_mode* is the residence mode to be associated with the designated objects.

  A24: Indicates that the designated objects must reside within 24 bit addressable virtual storage (that is, below the 16 megabyte virtual storage line under MVS/XA).

  ANY: Indicates that the designated objects may reside anywhere in virtual storage (that is, either above or below the 16 megabyte virtual storage line under MVS/XA).

On non-extended architecture machines the pragma is effectively ignored, since only 16 megabytes of virtual address space are available and all virtual addresses implicitly meet the A24 residence mode criteria.

Under MVS/XA the pragma is significant for data whose residence mode must be explicitly controlled, e.g. data which is to be passed to non-Ada code via the pragma INTERFACE.

In the absence of the pragma RMODE, the default residence mode associated with the objects designated by an access type is ANY.

The *access_type_name* must be a simple name. The pragma RMODE and the access type declaration to which it refers must both occur immediately within the same declarative part, package specification or task specification; the declaration must occur before the pragma.

1.8 MAP_TASK

Pragma MAP_TASK controls the mapping of Ada tasks to operating system processes. The pragma refers to a set of tasks of the same task type, all instances of which will be mapped in the same manner.

In the case of a task specification including the reserved word **type**, the declaration defines a task type. The set of tasks represented by such a task type name comprises all task objects of the specified type.

In the case of a task specification without the reserved word **type**, the declaration is considered to introduce an anonymous task type with a single instance [9.1]. The set of tasks represented by such an anonymous task type name contains exactly this one task.

This pragma takes the form:

pragma MAP_TASK (*task_type_name*);

where:

- *task_type_name* is the name of the task or task type.

Under CMS the pragma is effectively ignored since no operating system processes exist.

Under MVS and MVS/XA the pragma controls the mapping of Ada tasks to MVS system processes. All instances of an Ada task type to which a pragma MAP_TASK applies are mapped to their own operating system processes. Such Ada tasks never share an operating system process.

In the absence of the pragma MAP_TASK, an Ada task is mapped to a default operating system process and internally scheduled, together with all other Ada tasks mapped to this process, by the Ada Run-Time Executive.

Pragma MAP_TASK is allowed in the same places as a declarative item and must refer to a task or task type declared by an earlier declarative item of the same declarative part or package specification.

1.9 Other Pragmas

Pragmas IMPROVE and PACK are discussed in detail in the section on representation clauses (Chapter 4).

Pragma PRIORITY is accepted with the range of priorities running from 1 to 10 (see the definition of the predefined package SYSTEM in Chapter 3). The undefined priority (no pragma PRIORITY) is treated as though it were less than any defined priority value.

In addition to pragma SUPPRESS, it is possible to suppress checks in a given compilation by the use of the Compiler option CHECKS.

The following language defined pragmas have no effect:

CONTROLLED
MEMORY_SIZE
OPTIMIZE
STORAGE_UNIT
SYSTEM_NAME

Note that all access types are implemented by default as controlled collections as described in [4.8] (see section 10.1).

For optimisations certain facilities are provided through the use of the COMPILE command with the option IMPROVE (see *User's Guide,* Chapter 4).

CHAPTER 2

Implementation-Dependent Attributes

This chapter describes the implementation-dependent attributes and limitations on the use of the attribute ADDRESS.

The following implementation-dependent attributes are provided for use in record representation clauses only (see Section 4.8):

C'OFFSET
R'RECORD_SIZE
R'VARIANT_INDEX
C'ARRAY_DESCRIPTOR
C'RECORD_DESCRIPTOR

where C is the name of a record component and R is the name of a record type.

The following are also implementation-dependent attributes and are described in the remainder of this chapter:

T'DESCRIPTOR_SIZE
T'IS_ARRAY
SYSTEM.ADDRESS'IMPORT

where T is the name of any type or subtype.

2.1 T'DESCRIPTOR_SIZE

For a prefix T that denotes a type or subtype, this attribute yields the size (in bits) required to hold a descriptor for an object of the type T, allocated on the heap or written to a file. If T is constrained, T'DESCRIPTOR_SIZE will yield the value 0.

2.2 T'IS_ARRAY

For a prefix T that denotes a type or subtype, this attribute yields the value TRUE if T denotes an array type or an array subtype; otherwise, it yields the value FALSE.

2.3 SYSTEM.ADDRESS'IMPORT

This attribute is a function which takes two strings as arguments; the first one denotes a language name and the second one denotes an external symbol name. It yields the address of this external symbol. The prefix of this attribute must be SYSTEM.ADDRESS. The value of this attribute is of the type SYSTEM.ADDRESS. The syntax is;

SYSTEM.ADDRESS'IMPORT ("*Language_name*", "*external_symbol_name*")

Following are two examples which illustrate the use of this attribute.

Example 1:

SYSTEM.ADDRESS'IMPORT is used in an address clause in order to access an assembler DSECT:

For the language ASSEMBLER:

```
            ENTRY ERRNO
MYDATA      DSECT
....
ERRNO       DS F
....
            END
```

For the language Ada:

```
package MY_PACK is

   ERROR_NO:LONG_INTEGER;
   for ERROR_NO use at SYSTEM.ADDRESS'IMPORT ("ASSEMBLER",
   "ERRNO");
   ...

end MY_PACK;
```

Note that impicit initializations are performed on the declaration of objects; objects of type access are implicitly initialized to null.

Example 2:

The second example shows another use of 'IMPORT which avoids implicit initializations.

SYSTEM.ADDRESS'IMPORT is used in a renaming declaration to give a new name to an external object:

For the language ASSEMBLER:

```
            ENTRY REC
REC         DSECT
I1          DS F
I2          DS F
            END
```

For the language Ada:

```
type RECORD_A is
    record
        I1:INTEGER;
        I2:INTEGER;
    end record;

type ACCESS_RECORD is access RECORD_A;
function COVERT_TO_ACCESS_RECORD is new UNCHECKED_CONVERSION
                                (SYSTEM.ADDRESS, ACCESS RECORD);
X:RECORD_A renames CONVERT_TO_ACCESS_RECORD
(SYSTEM.ADDRESS'IMPORT ("ASSEMBLER, "REC")).all;
```

In this example, no implicit initialization is done on the renamed object X.

Note that the object is actually defined in the external world and is only *referenced* in the Ada world.

2.4 Limitations on the use of the attribute ADDRESS

The attribute ADDRESS is implemented for all prefixes that have meaningful addresses. The following entities do not have meaningful addresses. The attribute ADDRESS will deliver the value SYSTEM.NULL_ADDRESS if applied to such prefixes and a compilation warning will be issued.

- A constant or named number that is implemented as an immediate value (i.e. does not have any space allocated for it).
- A package specification that is not a library unit.
- A package body that is not a library unit or subunit.
- A package body that is not a library unit or subunit

CHAPTER 3

Specification of the Package SYSTEM

```
package SYSTEM is

   type NAME is (I80X86,
                 I80386,
                 MC680X0,
                 S370,
                 TRANSPUTER,
                 VAX);

   SYSTEM_NAME  : constant NAME := S370;

   STORAGE_UNIT : constant := 8;
   MAX_INT      : constant := 2**31 - 1;
   MIN_INT      : constant := - (2**31);
   MAX_MANTISSA : constant := 31;
   FINE_DELTA   : constant := 2#1.0#E-31;
   MAX_DIGITS   : constant := 18;
   MEMORY_SIZE  : constant := 2**32;
   TICK         : constant := 0.01;

   subtype PRIORITY is INTEGER range 1 .. 10;

   type ADDRESS is private;
   NULL_ADDRESS : constant ADDRESS;


   function VALUE (LEFT : in STRING) return ADDRESS;

   subtype ADDRESS_STRING is STRING(1..8);

   function IMAGE (LEFT : in ADDRESS) return ADDRESS_STRING;


   type OFFSET is range -(2**31) .. 2**31-1;
   -- This type is used to measure a number of storage units (bytes).

   function SAME_SEGMENT (LEFT, RIGHT : in ADDRESS) return BOOLEAN;

   ADDRESS_ERROR : exception;
```

```
   function "+" (LEFT : in ADDRESS; RIGHT : in OFFSET)  return ADDRESS;
   function "+" (LEFT : in OFFSET;  RIGHT : in ADDRESS) return ADDRESS;
   function "-" (LEFT : in ADDRESS; RIGHT : in OFFSET)  return ADDRESS;

   function "-" (LEFT : in ADDRESS; RIGHT : in ADDRESS) return OFFSET;

   function "<=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
   function "<"  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
   function ">=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
   function ">"  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;

   function "mod" (LEFT : in ADDRESS; RIGHT : in POSITIVE) return NATURAL;


   type ROUND_DIRECTION is (DOWN, UP);

   function ROUND (VALUE     : in ADDRESS;
                   DIRECTION : in ROUND_DIRECTION;
                   MODULUS   : in POSITIVE) return ADDRESS;


   generic
      type TARGET is private;
   function FETCH_FROM_ADDRESS (A : in ADDRESS) return TARGET;
   generic
      type TARGET is private;
   procedure ASSIGN_TO_ADDRESS (A : in ADDRESS; T : in TARGET);
   -- These routines are provided to perform READ/WRITE operations in memory.


   type OBJECT_LENGTH is range 0 .. 2**31 -1;
   -- This type is used to designate the size of an object in storage units.

   procedure MOVE (TO     : in ADDRESS;
                   FROM   : in ADDRESS;
                   LENGTH : in OBJECT_LENGTH);

end SYSTEM;
```

The function VALUE may be used to convert a string into an address. The string is a sequence of up to eight hexadecimal characters (digits or letters in upper or lower case in the range A..F) representing a virtual address. The exception CONSTRAINT_ERROR is raised if the string does not have the proper syntax.

The function IMAGE may be used to convert an address to a string which is a sequence of exactly eight hexadecimal digits, using the characters 0..9 and A..F.

The function SAME_SEGMENT always returns TRUE and the exception ADDRESS_ERROR is never raised as the 370 is a non segmented architecture.

The functions "+" and "-" with an ADDRESS and an OFFSET parameter provide support to perform address computations. The OFFSET parameter is added to, or subtracted from the address. The exception CONSTRAINT_ERROR can be raised by these functions.

The function "-" with the two ADDRESS parameters may be used to return the distance between the specified addresses.

The functions "<=", "<", ">=" and ">" may be used to perform a comparison on the specified addresses. The comparison is signed.

The function "mod" may be used to return the offset of LEFT address relative to the memory block immediately below it starting at a multiple of RIGHT storage units.

The function ROUND may be used to return the specified address rounded to a specific value in a particular direction.

The generic function FETCH_FROM_ADDRESS may be used to read data objects from given addresses in store. The generic function ASSIGN_TO_ADDRESS may be used to write data objects to given addresses in store. These routines may not be instantiated with unconstrained types.

The procedure MOVE may be used to copy LENGTH storage units starting at the address FROM to the address TO. The source and destination locations may overlap.

CHAPTER 4

Restrictions on Representation Clauses

This section explains how objects are represented and allocated by the Alsys IBM 370 Ada Compiler and how it is possible to control this using representation clauses.

The representation of an object is closely connected with its type. For this reason this section addresses successively the representation of enumeration, integer, floating point, fixed point, access, task, array and record types. For each class of type the representation of the corresponding objects is described.

Except in the case of array and record types, the description of each class of type is independent of the others. To understand the representation of an array type it is necessary to understand first the representation of its components. The same rule applies to a record type.

Apart from implementation defined pragmas, Ada provides three means to control the size of objects:

- a (predefined) pragma PACK, when the object is an array, an array component, a record or a record component
- a record representation clause, when the object is a record or a record component
- a size specification, in any case.

For each class of types the effect of a size specification is described. Interaction between size specifications, packing and record representation clauses is described under array and record types.

4.1 Enumeration Types

Internal codes of enumeration literals

When no enumeration representation clause applies to an enumeration type, the internal code associated with an enumeration literal is the position number of the enumeration literal. Then, for an enumeration type with n elements, the internal codes are the integers 0, 1, 2, ... , n-1.

An enumeration representation clause can be provided to specify the value of each internal code as described in [13.3]. The Alsys Compiler fully implements enumeration representation clauses.

As internal codes must be machine integers the internal codes provided by an enumeration representation clause must be in the range $-2^{31} .. 2^{31}-1$.

Encoding of enumeration values

An enumeration value is always represented by its internal code in the program generated by the Compiler.

Enumeration subtypes

Minimum size: The minimum size of an enumeration subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the values of the internal codes associated with the first and last enumeration values of the subtype, then its minimum size L is determined as follows. For $m >= 0$, L is the smallest positive integer such that $M <= 2^L-1$. For $m < 0$, L is the smallest positive integer such that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$.

For example:

```
type COLOR is (GREEN, BLACK, WHITE, RED, BLUE, YELLOW);
-- The minimum size of COLOR is 3 bits.

subtype BLACK_AND_WHITE is COLOR range BLACK .. WHITE;
-- The minimum size of BLACK_AND_WHITE is 2 bits.
```

```
subtype BLACK_OR_WHITE is BLACK_AND_WHITE range X .. X;
-- Assuming that X is not static, the minimum size of BLACK_OR_WHITE is
-- 2 bits (the same as the minimum size of the static type mark
-- BLACK_AND_WHITE).
```

Size: When no size specification is applied to an enumeration type or first named subtype, the objects of that type or first named subtype are represented as signed integers if the internal code associated with the first enumeration value is negative, and as unsigned integers otherwise. The machine provides 8, 16 and 32 bit integers, and the Compiler selects automatically the smallest machine integer which can hold each of the internal codes of the enumeration type (or subtype). The size of the enumeration type and of any of its subtypes is thus 8, 16 or 32 bits.

When a size specification is applied to an enumeration type, this enumeration type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

For example:

```
type EXTENDED is
 ( -- The usual American ASCII characters.
      NUL,    SOH,    STX,    ETX,    EOT,    ENQ,    ACK,    BEL,
      BS,     HT,     LF,     VT,     FF,     CR,     SO,     SI,
      DLE,    DC1,    DC2,    DC3,    DC4,    NAK,    SYN,    ETB,
      CAN,    EM,     SUB,    ESC,    FS,     GS,     RS,     US,
      ' ',    '!',    '"',    '#',    '$',    '%',    '&',    ''',
      '(',    ')',    '*',    '+',    ',',    '-',    '.',    '/',
      '0',    '1',    '2',    '3',    '4',    '5',    '6',    '7',
      '8',    '9',    ':',    ';',    '<',    '=',    '>',    '?',
      '@',    'A',    'B',    'C',    'D',    'E',    'F',    'G',
      'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',
      'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',
      'X',    'Y',    'Z',    '[',    '\',    ']',    '^',    '_',
      '`',    'a',    'b',    'c',    'd',    'e',    'f',    'g',
      'h',    'i',    'j',    'k',    'l',    'm',    'n',    'o',
      'p',    'q',    'r',    's',    't',    'u',    'v',    'w',
      'x',    'y',    'z',    '{',    '|',    '}',    '~',    DEL,
```

```
        UPPER_ARROW,
        LOWER_ARROW,
        UPPER_LEFT_CORNER,
        UPPER_RIGHT_CORNER,
        LOWER_RIGHT_CORNER,
        LOWER_LEFT_CORNER,
        ...);

    for EXTENDED'SIZE use 8;
    -- The size of type EXTENDED will be one byte. Its objects will be represented
    -- as unsigned 8 bit integers.
```

The Alsys Compiler fully implements size specifications. Nevertheless, as enumeration values are coded using integers, the specified length cannot be greater than 32 bits.

Object size: Provided its size is not constrained by a record component clause or a pragma PACK, an object of an enumeration subtype has the same size as its subtype.

Alignment: An enumeration subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, halfword aligned if the size of the subtype is less than or equal to 16 bits and word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an enumeration subtype is a multiple of the alignment of the corresponding subtype.

4.2 Integer Types

Predefined integer types

There are three predefined integer types in the Alsys implementation for IBM 370 machines:

```
type SHORT_SHORT_INTEGER is range -2**07 .. 2**07-1;
type SHORT_INTEGER       is range -2**15 .. 2**15-1;
type INTEGER             is range -2**31 .. 2**31-1;
```

Selection of the parent of an integer type

An integer type declared by a declaration of the form:

```
type T is range L .. R;
```

is implicitly derived from either the SHORT_INTEGER or INTEGER predefined integer type. The Compiler automatically selects the predefined integer type whose range is the shortest that contains the values L to R inclusive. Note that the SHORT_SHORT_INTEGER representation is never automatically selected by the Compiler.

Encoding of integer values

Binary code is used to represent integer values, using a conventional two's complement representation.

Integer subtypes

Minimum size: The minimum size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form (that is to say, in an unbiased form which includes a sign bit only if the range of the subtype includes negative values).

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the lower and upper bounds of the subtype, then its minimum size L is determined as follows. For m >= 0, L is the smallest positive integer such that M <= 2^L-1. For m < 0, L is the smallest positive integer such that -2^{L-1} <= m and M <= $2^{L-1}-1$.

For example:

```
subtype S is INTEGER range 0 .. 7;
-- The minimum size of S is 3 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of
-- D is 3 bits (the same as the minimum size of the static type mark S).
```

Size: The sizes of the predefined integer types SHORT_SHORT_INTEGER, SHORT_INTEGER and INTEGER are respectively 8, 16 and 32 bits.

When no size specification is applied to an integer type or to its first named subtype (if any), its size and the size of any of its subtypes is the size of the predefined type from which it derives, directly or indirectly.

For example:

```
type S is range 80 .. 100;
-- S is derived from SHORT_INTEGER, its size is 16 bits.

type J is range 0 .. 65535;
-- J is derived from INTEGER, its size is 32 bits.

type N is new J range 80 .. 100;
-- N is indirectly derived from INTEGER, its size is 32 bits.
```

When a size specification is applied to an integer type, this integer type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

For example:

```
type S is range 80 .. 100;
for S'SIZE use 32;
-- S is derived from SHORT_INTEGER, but its size is 32 bits
-- because of the size specification.
```

```
type J is range 0 .. 255;
for J'SIZE use 8;
-- J is derived from SHORT_INTEGER, but its size is 8 bits because
-- of the size specification.

type N is new J range 80 .. 100;
-- N is indirectly derived from SHORT_INTEGER, but its size is 8 bits
-- because N inherits the size specification of J.
```

The Alsys Compiler fully implements size specifications. Nevertheless, as integers are implemented using machine integers, the specified length cannot be greater than 32 bits.

Object size: Provided its size is not constrained by a record component clause or a pragma PACK, an object of an integer subtype has the same size as its subtype.

Alignment: An integer subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, halfword aligned if the size of the subtype is less than or equal to 16 bits and word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an integer subtype is a multiple of the alignment of the corresponding subtype.

4.3 Floating Point Types

Predefined floating point types

There are three predefined floating point types in the Alsys implementation for IBM 370 machines:

```
type SHORT_FLOAT is
   digits 6 range -2.0**252*(1.0-2.0**-24) .. 2.0**252*(1.0-2.0**-24);

type FLOAT is
   digits 15 range -2.0**252*(1.0-2.0**-56) .. 2.0**252*(1.0-2.0**-56);

type LONG_FLOAT is
   digits 18 range -2.0**252*(1.0-2.0**-112) .. 2.0**252*(1.0-2.0**-112);
```

Selection of the parent of a floating point type

A floating point type declared by a declaration of the form:

```
type T is digits D [range L .. R];
```

is implicitly derived from a predefined floating point type. The Compiler automatically selects the smallest predefined floating point type whose number of digits is greater than or equal to D and which contains the values L and R.

Encoding of floating point values

In the program generated by the Compiler, floating point values are represented using the IBM 370 data formats for single precision, double precision and extended precision floating point values as appropriate.

Values of the predefined type SHORT_FLOAT are represented using the single precision format, values of the predefined type FLOAT are represented using the double precision format and values of the predefined type LONG_FLOAT are represented using the extended precision format. The values of any other floating point type are represented in the same way as the values of the predefined type from which it derives, directly or indirectly.

Floating point subtypes

Minimum size: The minimum size of a floating point subtype is 32 bits if its base type is SHORT_FLOAT or a type derived from SHORT_FLOAT, 64 bits if its base type is FLOAT or a type derived from FLOAT and 128 bits if its base type is LONG_FLOAT or a type derived from LONG_FLOAT.

Size: The sizes of the predefined floating point types SHORT_FLOAT, FLOAT and LONG_FLOAT are respectively 32, 64 and 128 bits.

The size of a floating point type and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

The only size that can be specified for a floating point type or first named subtype using a size specification is its usual size (32, 64 or 128 bits).

Object size: An object of a floating point subtype has the same size as its subtype.

Alignment: A floating point subtype is word aligned if its size is 32 bits and double word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a floating point subtype is a multiple of the alignment of the corresponding subtype.

4.4 Fixed Point Types

Small of a fixed point type

If no specification of small applies to a fixed point type, then the value of small is determined by the value of delta as defined by [3.5.9].

A specification of small can be used to impose a value of small. The value of small is required to be a power of two.

Predefined fixed point types

To implement fixed point types, the Alsys Compiler for IBM 370 machines uses a set of anonymous predefined types of the form:

```
type FIXED is delta D range (-2**15)*S .. (2**15-1)*S;
for FIXED'SMALL use S;

type LONG_FIXED is delta D range (-2**31)*S .. (2**31-1)*S;
for LONG_FIXED'SMALL use S;
```

where D is any real value and S any power of two less than or equal to D.

Selection of the parent of a fixed point type

A fixed point type declared by a declaration of the form:

```
type T is delta D range L .. R;
```

possibly with a small specification:

```
for T'SMALL use S;
```

is implicitly derived from a predefined fixed point type. The Compiler automatically selects the predefined fixed point type whose small and delta are the same as the small and delta of T and whose range is the shortest that includes the values L and R.

Encoding of fixed point values

In the program generated by the Compiler, a safe value V of a fixed point subtype F is represented as the integer:

V / F'BASE'SMALL

Fixed point subtypes

Minimum size: The minimum size of a fixed point subtype is the minimum number of binary digits that is necessary for representing the values of the range of the subtype using the small of the base type (that is to say, in an unbiased form which includes a sign bit only if the range of the subtype includes negative values).

For a static subtype, if it has a null range its minimum size is 1. Otherwise, s and S being the bounds of the subtype, if i and I are the integer representations of m and M, the smallest and the greatest model numbers of the base type such that s < m and M < S, then the minimum size L is determined as follows. For i >= 0, L is the smallest positive integer such that I <= $2^{L}-1$. For i < 0, L is the smallest positive integer such that -2^{L-1} <= i and I <= $2^{L-1}-1$.

For example:

```
type F is delta 2.0 range 0.0 .. 500.0;
-- The minimum size of F is 8 bits.

subtype S is F delta 16.0 range 0.0 .. 250.0;
-- The minimum size of S is 7 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of D is 7 bits
-- (the same as the minimum size of its type mark S).
```

Size: The sizes of the sets of predefined fixed point types FIXED and LONG_FIXED are 16 and 32 bits respectively.

When no size specification is applied to a fixed point type or to its first named subtype, its size and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

For example:

```
type F is delta 0.01 range 0.0 .. 2.0;
-- F is derived from a 16 bit predefined fixed type, its size is 16 bits.

type L is delta 0.01 range 0.0 .. 300.0;
-- L is derived from a 32 bit predefined fixed type, its size is 32 bits.

type N is new L range 0.0 .. 2.0;
-- N is indirectly derived from a 32 bit predefined fixed type, its size is 32 bits.
```

When a size specification is applied to a fixed point type, this fixed point type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

For example:

```
type F is delta 0.01 range 0.0 .. 2.0;
for F'SIZE use 32;
-- F is derived from a 16 bit predefined fixed type, but its size is 32 bits
-- because of the size specification.

type L is delta 0.01 range 0.0 .. 300.0;
for F'SIZE use 16;
-- F is derived from a 32 bit predefined fixed type, but its size is 16 bits
-- because of the size specification.
-- The size specification is legal since the range contains no negative values
-- and therefore no sign bit is required.

type N is new F range 0.8 .. 1.0;
-- N is indirectly derived from a 16 bit predefined fixed type, but its size is
-- 32 bits because N inherits the size specification of F.
```

The Alsys Compiler fully implements size specifications. Nevertheless, as fixed point objects are represented using machine integers, the specified length cannot be greater than 32 bits.

Object size: Provided its size is not constrained by a record component clause or a pragma PACK, an object of a fixed point type has the same size as its subtype.

Alignment: A fixed point subtype is byte aligned if its size is less than or equal to 8 bits, halfword aligned if the size of the subtype is less than or equal to 16 bits and word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a fixed point subtype is a multiple of the alignment of the corresponding subtype.

4.5 Access Types

Collection Size

When no specification of collection size applies to an access type, no storage space is reserved for its collection, and the value of the attribute STORAGE_SIZE is then 0.

As described in [13.2], a specification of collection size can be provided in order to reserve storage space for the collection of an access type. The Alsys Compiler fully implements this kind of specification.

Encoding of access values

Access values are machine addresses represented as 32 bit values. The implementation uses the top (most significant) bit of such a 32 bit value to pass additional information to the Ada Run-Time Executive.

Access subtypes

Minimum size: The minimum size of an access subtype is 32 bits.

Size: The size of an access subtype is 32 bits, the same as its minimum size.

The only size that can be specified for an access type using a size specification is its usual size (32 bits).

Object size: An object of an access subtype has the same size as its subtype, thus an object of an access subtype is always 32 bits long.

Alignment: An access subtype is always word aligned.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an access subtype is always on a word boundary, since its subtype is word aligned.

4.6 Task Types

Storage for a task activation

When no length clause is used to specify the storage space to be reserved for a task activation, the storage space indicated at bind time is used for this activation.

As described in [13.2], a length clause can be used to specify the storage space for the activation of each of the tasks of a given type. In this case the value indicated at bind time is ignored for this task type, and the length clause is obeyed.

It is not allowed to apply such a length clause to a derived type. The same storage space is reserved for the activation of a task of a derived type as for the activation of a task of the parent type.

Encoding of task values

Task values are machine addresses.

Task subtypes

Minimum size: The minimum size of a task subtype is 32 bits.

Size: The size of a task subtype is 32 bits, the same as its minimum size.

The only size that can be specified for a task type using a size specification is its usual size (32 bits).

Object size: An object of a task subtype has the same size as its subtype. Thus an object of a task subtype is always 32 bits long.

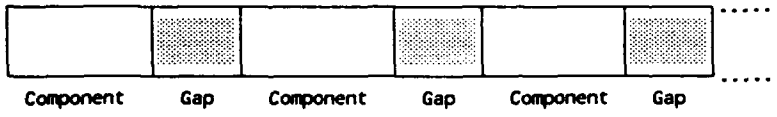
Alignment: A task subtype is always word aligned.

Object address: Provided its alignment is not constrained by a record representation clause, the address of an object of a task subtype is always on a word boundary, since its subtype is word aligned.

4.7 Array Types

Layout of an array

Each array is allocated in a contiguous area of storage units. All the components have the same size. A gap may exist between two consecutive components (and after the last one). All the gaps have the same size.



Components

If the array is not packed, the size of the components is the size of the subtype of the components.

For example:

```
type A is array (1 .. 8) of BOOLEAN;
-- The size of the components of A is the size of the type BOOLEAN: 8 bits.

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 4;
type BINARY_CODED_DECIMAL is
      array (INTEGER range <>) of DECIMAL_DIGIT;
-- The size of the type DECIMAL_DIGIT is 4 bits. Thus in an array of
-- type BINARY_CODED_DECIMAL each component will be represented in
-- 4 bits as in the usual BCD representation.
```

If the array is packed and its components are neither records nor arrays, the size of the components is the minimum size of the subtype of the components.

For example:

```
type A is array (1 .. 8) of BOOLEAN;
pragma PACK(A);
-- The size of the components of A is the minimum size of the type BOOLEAN:
-- 1 bit.
```

```
type DECIMAL_DIGIT is range 0 .. 9;
type BINARY_CODED_DECIMAL is
     array (INTEGER range <>) of DECIMAL_DIGIT;
pragma PACK(BINARY_CODED_DECIMAL);
-- The size of the type DECIMAL_DIGIT is 16 bits, but, as
-- BINARY_CODED_DECIMAL is packed, each component of an array of this
-- type will be represented in 4 bits as in the usual BCD representation.
```

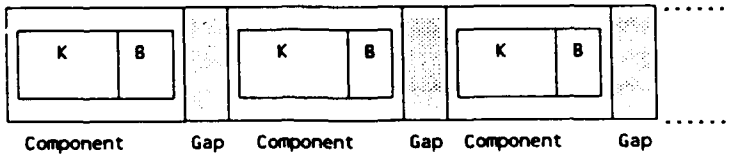
Packing the array has no effect on the size of the components when the components are records or arrays.

Gaps

If the components are records or arrays, no size specification applies to the subtype of the components and the array is not packed, then the Compiler may choose a representation with a gap after each component; the aim of the insertion of such gaps is to optimize access to the array components and to their subcomponents. The size of the gap is chosen so that the relative displacement of consecutive components is a multiple of the alignment of the subtype of the components. This strategy allows each component and subcomponent to have an address consistent with the alignment of its subtype

For example:

```
type R is
     record
          K : INTEGER; -- INTEGER is word aligned.
          B : BOOLEAN; -- BOOLEAN is byte aligned.
     end record;
     -- Record type R is word aligned. Its size is 40 bits.

type A is array (1 .. 10) of R;
-- A gap of three bytes is inserted  after each component in order to respect the
-- alignment of type R. The size of an array of type A will be 640 bits.
```

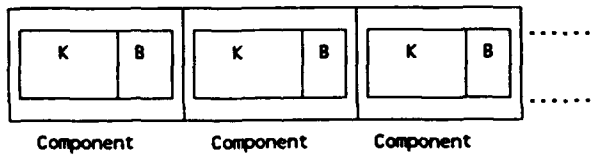


Array of type A: each subcomponent K has a word offset.

If a size specification applies to the subtype of the components or if the array is packed, no gaps are inserted.

For example:

```
type R is
    record
        K : INTEGER;
        B : BOOLEAN;
    end record;

type A is array (1 .. 10) of R;
pragma PACK(A);
-- There is no gap in an array of type A because A is packed.
-- The size of an object of type A will be 400 bits.

type NR is new R;
for NR'SIZE use 40;

type B is array (1 .. 10) of NR;
-- There is no gap in an array of type B because NR has a size specification.
-- The size of an object of type B will be 400 bits.
```



Array of type A or B: a subcomponent K can have any byte offset.

Array subtypes

Size: The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of the gaps (if any). If the subtype is unconstrained, the maximum number of components is considered.

The size of an array subtype cannot be computed at compile time

- if it has non-static constraints or is an unconstrained array type with non-static index subtypes (because the number of components can then only be determined at run time).
- if the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).

As has been indicated above, the effect of a pragma PACK on an array type is to suppress the gaps and to reduce the size of the components. The consequence of packing an array type is thus to reduce its size.

If the components of an array are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static, the Compiler ignores any pragma PACK applied to the array type but issues a warning message. Apart from this limitation, array packing is fully implemented by the Alsys Compiler.

The only size that can be specified for an array type or first named subtype using a size specification is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of an array is as expected by the application.

Object size: The size of an object of an array subtype is always equal to the size of the subtype of the object.

Alignment: If no pragma PACK applies to an array subtype and no size specification applies to its components, the array subtype has the same alignment as the subtype of its components.

If a pragma PACK applies to an array subtype or if a size specification applies to its components (so that there are no gaps), the alignment of the array subtype is the lesser of the alignment of the subtype of its components and the relative displacement of the components.

Object address: Provided its alignment is not constrained by a record representation clause, the address of an object of an array subtype is a multiple of the alignment of the corresponding subtype.

4.8 Record Types

Layout of a record

Each record is allocated in a contiguous area of storage units. The size of a record component depends on its type. Gaps may exist between some components.

The positions and the sizes of the components of a record type object can be controlled using a record representation clause as described in [13.4]. In the Alsys implementation for IBM 370 machines there is no restriction on the position that can be specified for a component of a record. Bits within a storage unit are numbered from 0 to 7, with the most-significant bit numbered 0. The range of bits specified in a component clause may extend into following storage units. If a component is not a record or an array, its size can be any size from the minimum size to the size of its subtype. If a component is a record or an array, its size must be the size of its subtype:

```
type ACCESS_KEY is range 0..15;
-- The size of ACCESS_KEY is 16 bits, the minimum size is 4 bits

type CONDITIONS is (ZERO, LESS_THAN, GREATER_THAN, OVERFLOW);
-- The size of CONDITIONS is 8 bits, the minimum size is 2 bits

type PROG_EXCEPTION is (FIX_OVFL, DEC_OVFL, EXP_UNDFL, SIGNIF);
type PROG_MASK is array (PROG_EXCEPTION) of BOOLEAN;
pragma PACK (PROG_MASK);
-- The size of PROG_MASK is 4 bits

type ADDRESS is range 0..2**24-1;
for ADDRESS'SIZE use 24;
-- ADDRESS represents a 24 bit memory address

type PSW is
    record
        PER_MASK      : BOOLEAN;
        DAT_MODE      : BOOLEAN;
        IO_MASK       : BOOLEAN;
        EXTERNAL_MASK : BOOLEAN;
        PSW_KEY       : ACCESS_KEY;
        EC_MODE       : BOOLEAN;
        MACHINE_CHECK : BOOLEAN
        WAIT_STATE    : BOOLEAN;
        PROBLEM_STATE : BOOLEAN;
```

```
        ADDRESS_SPACE   : BOOLEAN;
        CONDITION_CODE: CONDITIONS;
        PROGRAM_MASK    : PROG_MASK;
        INSTR_ADDRESS   : ADDRESS;
    end record;

-- This type can be used to map the program status word of the IBM 370

for PSW use
    record at mod 8;
        PER_MASK        at 0    range 1..1;
        DAT_MODE        at 0    range 5..5;
        IO_MASK         at 0    range 6..6;
        EXTERNAL_MASK   at 0    range 7..7;
        PSW_KEY         at 1    range 0..3;
        EC_MODE         at 1    range 4..4;
        MACHINE_CHECK   at 1    range 5..5;
        WAIT_STATE      at 1    range 6..6;
        PROBLEM_STATE   at 1    range 7..7;
        ADDRESS_SPACE   at 2    range 0..0;
        CONDITION_CODEat 2      range 2..3;
        PROGRAM_MASK    at 2    range 4..7;
        INSTR_ADDRESS   at 5    range 0..23;
    end record;
```

A record representation clause need not specify the position and the size for every component.

If no component clause applies to a component of a record, its size is the size of its subtype. Its position is chosen by the Compiler so as to optimize access to the components of the record: the offset of the component is chosen as a multiple of the alignment of the component subtype. Moreover, the Compiler chooses the position of the component so as to reduce the number of gaps and thus the size of the record objects.

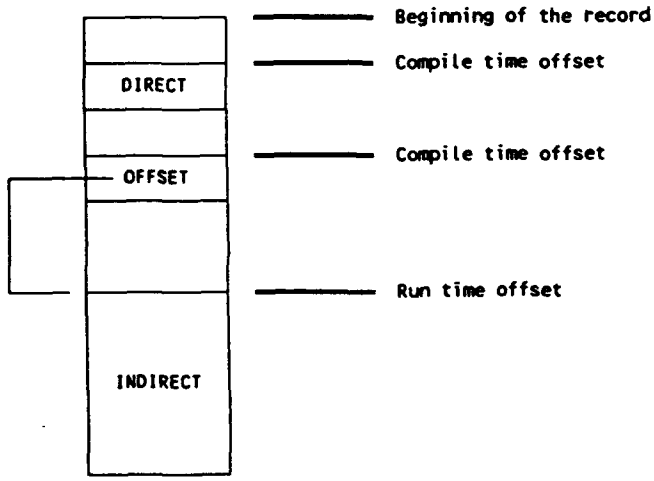
Because of these optimisations, there is no connection between the order of the components in a record type declaration and the positions chosen by the Compiler for the components in a record object.

Pragma PACK has no further effect on records. The Alsys Compiler always optimizes the layout of records as described above.

In the current version, it is not possible to apply a record representation clause to a derived type. The same storage representation is used for an object of a derived type as for an object of the parent type.

Indirect components

If the offset of a component cannot be computed at compile time, this offset is stored in the record objects at run time and used to access the component. Such a component is said to be indirect while other components are said to be direct:



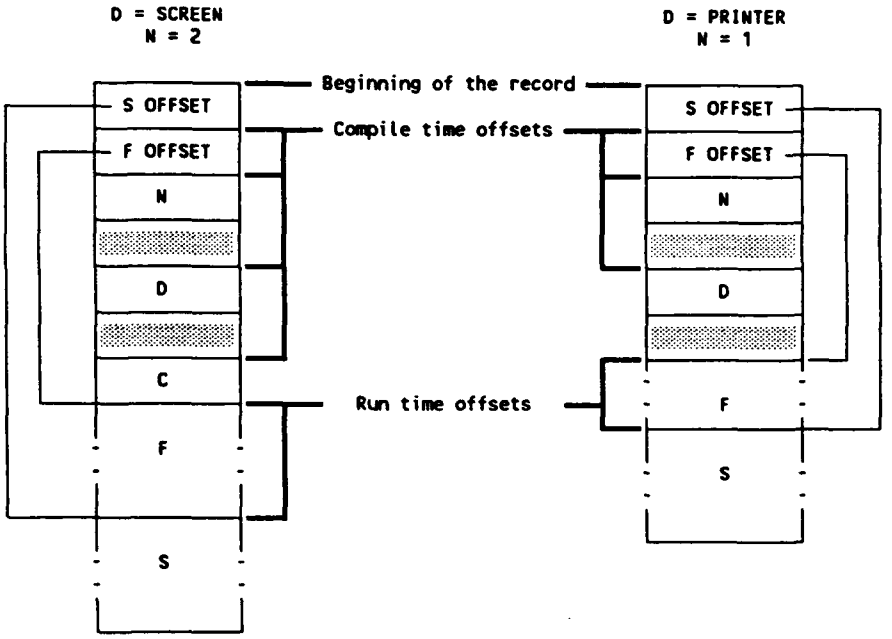
A direct and an indirect component

If a record component is a record or an array, the size of its subtype may be evaluated at run time and may even depend on the discriminants of the record. We will call these components dynamic components.

For example:

```
type DEVICE is (SCREEN, PRINTER);

type COLOR is (GREEN, RED, BLUE);

type SERIES is array (POSITIVE range < >) of INTEGER;

type GRAPH (L : NATURAL) is
    record
        X : SERIES(1 .. L); -- The size of X depends on L
        Y : SERIES(1 .. L); -- The size of Y depends on L
    end record;

Q : POSITIVE;

type PICTURE (N : NATURAL; D : DEVICE) is
    record
        F : GRAPH(N); -- The size of F depends on N
        S : GRAPH(Q); -- The size of S depends on Q
        case D is
            when SCREEN =>
               C : COLOR;
            when PRINTER =>
               null;
        end case;
    end record;
```

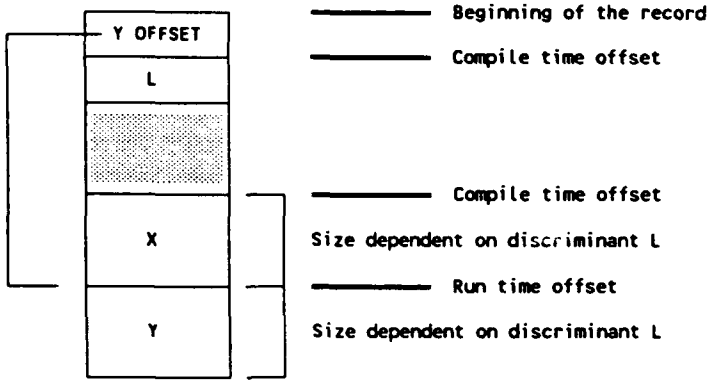
Any component placed after a dynamic component has an offset which cannot be evaluated at compile time and is thus indirect. In order to minimize the number of indirect components, the Compiler groups the dynamic components together and places them at the end of the record (see diagram on the following page):



The record type PICTURE: F and S are placed at the end of the record

As a result of this strategy, the only indirect components are dynamic components. However, not all dynamic components are necessarily indirect. If there are dynamic components in a component list which is not followed by a variant part, then exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time.

For example:



The record type GRAPH: the dynamic component X is a direct component.

The offset of an indirect component is always expressed in storage units.

The space reserved for the offset of an indirect component must be large enough to store the size of any value of the record type (the maximum potential offset). The Compiler evaluates an upper bound MS of this size and treats an offset as a component having an anonymous integer type whose range is 0 .. MS.

If C is the name of an indirect component, then the offset of this component can be denoted in a component clause by the implementation generated name C'OFFSET.

Implicit components

In some circumstances, access to an object of a record type or to its components involves computing information which only depends on the discriminant values. To avoid unnecessary recomputation, the Compiler stores this information in the record objects, updates it when the values of the discriminants are modified and uses it when the objects or their components are accessed. This information is stored in special components called implicit components.

An implicit component may contain information which is used when the record object or several of its components are accessed. In this case the component will be included in any record object (the implicit component is considered to be declared before any variant part in the record type declaration). There can be two components of this kind; one is called RECORD_SIZE and the other VARIANT_INDEX.

On the other hand an implicit component may be used to access a given record component. In this case the implicit component exists whenever the record component exists (the implicit component is considered to be declared at the same place as the record component). Components of this kind are called ARRAY_DESCRIPTORs or RECORD_DESCRIPTORs.

RECORD_SIZE

This implicit component is created by the Compiler when the record type has a variant part and its discriminants are defaulted. It contains the size of the storage space necessary to store the current value of the record object (note that the storage effectively allocated for the record object may be more than this).

The value of a RECORD_SIZE component may denote a number of bits or a number of storage units. In general it denotes a number of storage units, but if any component clause specifies that a component of the record type has an offset or a size which cannot be expressed using storage units, then the value designates a number of bits.

The implicit component RECORD_SIZE must be large enough to store the maximum size of any value of the record type. The Compiler evaluates an upper bound MS of this size and then considers the implicit component as having an anonymous integer type whose range is 0 .. MS.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'RECORD_SIZE.

VARIANT_INDEX

This implicit component is created by the Compiler when the record type has a variant part. It indicates the set of components that are present in a record value. It is used when a discriminant check is to be done.

Component lists that do not contain a variant part are numbered. These numbers are the possible values of the implicit component VARIANT_INDEX.

For example:

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

type DESCRIPTION (KIND : VEHICLE := CAR) is
    record
        SPEED : INTEGER;
        case KIND is
            when AIRCRAFT | CAR =>
              WHEELS : INTEGER;
              case KIND is
                when AIRCRAFT =>    -- 1
                  WINGSPAN : INTEGER;
                when others =>      -- 2
                  null;
              end case;
            when BOAT =>            -- 3
              STEAM  : BOOLEAN;
            when ROCKET =>          -- 4
              STAGES : INTEGER;
        end case;
    end record;
```

The value of the variant index indicates the set of components that are present in a record value:

Variant Index	Set
1	(KIND, SPEED, WHEELS, WINGSPAN)
2	(KIND, SPEED, WHEELS)
3	(KIND, SPEED, STEAM)
4	(KIND, SPEED, STAGES)

A comparison between the variant index of a record value and the bounds of an interval is enough to check that a given component is present in the value:

Component	Interval
KIND	--
SPEED	--
WHEELS	1 .. 2
WINGSPAN	1 .. 1
STEAM	3 .. 3
STAGES	4 .. 4

The implicit component VARIANT_INDEX must be large enough to store the number V of component lists that don't contain variant parts. The Compiler treats this implicit component as having an anonymous integer type whose range is 1 .. V.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'VARIANT_INDEX.

ARRAY_DESCRIPTOR

An implicit component of this kind is associated by the Compiler with each record component whose subtype is an anonymous array subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind ARRAY_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the ASSEMBLY parameter in the COMPILE command.

The Compiler treats an implicit component of the kind ARRAY_DESCRIPTOR as having an anonymous record type. If C is the name of the record component whose subtype is described by the array descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'ARRAY_DESCRIPTOR.

RECORD_DESCRIPTOR

An implicit component of this kind is associated by the Compiler with each record component whose subtype is an anonymous record subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind RECORD_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the ASSEMBLY parameter in the COMPILE command.

The Compiler treats an implicit component of the kind RECORD_DESCRIPTOR as having an anonymous record type. If C is the name of the record component whose subtype is described by the record descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'RECORD_DESCRIPTOR.

Suppression of implicit components

The Alsys implementation provides the capability of suppressing the implicit components RECORD_SIZE and/or VARIANT_INDEX from a record type. This can be done using an implementation defined pragma called IMPROVE. The syntax of this pragma is as follows:

pragma IMPROVE (TIME | SPACE , [ON =>] *simple_name*);

The first argument specifies whether TIME or SPACE is the primary criterion for the choice of the representation of the record type that is denoted by the second argument.

If TIME is specified, the Compiler inserts implicit components as described above. If on the other hand SPACE is specified, the Compiler only inserts a VARIANT_INDEX or a RECORD_SIZE component if this component appears in a record representation clause that applies to the record type. A record representation clause can thus be used to keep one implicit component while suppressing the other.

A pragma IMPROVE that applies to a given record type can occur anywhere that a representation clause is allowed for this type.

Record subtypes

Size: Unless a component clause specifies that a component of a record type has an offset or a size which cannot be expressed using storage units, the size of a record subtype is rounded up to a whole number of storage units.

The size of a constrained record subtype is obtained by adding the sizes of its components and the sizes of its gaps (if any). This size is not computed at compile time

- when the record subtype has non-static constraints,
- when a component is an array or a record and its size is not computed at compile time.

The size of an unconstrained record subtype is obtained by adding the sizes of the components and the sizes of the gaps (if any) of its largest variant. If the size of a component or of a gap cannot be evaluated exactly at compile time, an upper bound of this size is used by the Compiler to compute the subtype size.

The only size that can be specified for a record type or first named subtype using a size specification is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of a record is as expected by the application.

Object size: An object of a constrained record subtype has the same size as its subtype.

An object of an unconstrained record subtype has the same size as its subtype if this size is less than or equal to 8 Kbyte. If the size of the subtype is greater than this, the object has the size necessary to store its current value; storage space is allocated and released as the discriminants of the record change.

Alignment: When no record representation clause applies to its base type, a record subtype has the same alignment as the component with the highest alignment requirement.

When a record representation clause that does not contain an alignment clause applies to its base type, a record subtype has the same alignment as the component with the highest alignment requirement which has not been overridden by its component clause.

When a record representation clause that contains an alignment clause applies to its base type, a record subtype has an alignment that obeys the alignment clause.

Object address: Provided its alignment is not constrained by a representation clause, the address of an object of a record subtype is a multiple of the alignment of the corresponding subtype.

CHAPTER 5

Conventions for Implementation-Generated Names

Special record components are introduced by the Compiler for certain record type definitions. Such record components are implementation-dependent; they are used by the Compiler to improve the quality of the generated code for certain operations on the record types. The existence of these components is established by the Compiler depending on implementation-dependent criteria. Attributes are defined for referring to them in record representation clauses. An error message is issued by the Compiler if the user refers to an implementation-dependent component that does not exist. If the implementation-dependent component exists, the Compiler checks that the storage location specified in the component clause is compatible with the treatment of this component and the storage locations of other components. An error message is issued if this check fails.

There are four such attributes:

T'RECORD_SIZE	For a prefix T that denotes a record type. This attribute refers to the record component introduced by the Compiler in a record to store the size of the record object. This component exists for objects of a record type with defaulted discriminants when the sizes of the record objects depend on the values of the discriminants.
T'VARIANT_INDEX	For a prefix T that denotes a record type. This attribute refers to the record component introduced by the Compiler in a record to assist in the efficient implementation of discriminant checks. This component exists for objects of a record type with variant type.
C'ARRAY_DESCRIPTOR	For a prefix C that denotes a record component of an array type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the Compiler in a record to store information on subtypes of components that depend on discriminants.

C'RECORD_DESCRIPTOR For a prefix C that denotes a record component of a record type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the Compiler in a record to store information on subtypes of components that depend on discriminants.

CHAPTER 6

Address Clauses

6.1 Address Clauses for Objects

An address clause can be used to specify an address for an object as described in [13.5]. When such a clause applies to an object no storage is allocated for it in the program generated by the Compiler. The program accesses the object using the address specified in the clause.

An address clause is not allowed for task objects, nor for unconstrained records whose maximum possible size is greater than 8 Kbytes.

6.2 Address Clauses for Program Units

Address clauses for program units are not implemented.

6.3 Address Clauses for Entries

Address clauses for entries are not implemented.

CHAPTER 7

Restrictions on Unchecked Conversions

Unconstrained arrays are not allowed as target types.

Unconstrained record types without defaulted discriminants are not allowed as target types.

If the source and the target types are each scalar or access types, the sizes of the objects of the source and target types must be equal. If a composite type is used either as the source type or as the target type this restriction on the size does not apply.

If the source and the target types are each of scalar or access type or if they are both of composite type, the effect of the function is to return the operand.

In other cases the effect of unchecked conversion can be considered as a copy:

- if an unchecked conversion is achieved of a scalar or access source type to a composite target type, the result of the function is a copy of the source operand; the result has the size of the source.
- if an unchecked conversion is achieved of a composite source type to a scalar or access target type, the result of the function is a copy of the source operand; the result has the size of the target.

CHAPTER 8

Input-Output Packages

The predefined input-output packages SEQUENTIAL_IO [14.2.3], DIRECT_IO [14.2.5], TEXT_IO [14.3.10] and IO_EXCEPTIONS [14.5] are implemented as described in the Language Reference Manual.

The package LOW_LEVEL_IO [14.6], which is concerned with low-level machine-dependent input-output, is not implemented.

8.1 NAME Parameter

8.1.1 VM/CMS

The NAME parameter supplied to the Ada procedures CREATE or OPEN [14.2.1] may represent a CMS file name, a DDNAME specified using a FILEDEF command or an in-store file name.

The syntax of the Ada NAME parameter for VM/CMS is as follows:

file_name ::= *cms_file_name* |
%ddname |
\store_file_name

CMS file name

The syntax of a CMS file name as specified in the Ada NAME parameter is as follows:

cms_file_name ::= *fn* [*ft* [*fm*]]

where

fn is the CMS filename
ft is the CMS filetype
fm is the CMS filemode

If either the filename or filetype exceeds 8 characters then it is truncated. As indicated above, the filetype and filemode fields are not mandatory components of the NAME parameter. If the filemode is omitted, it defaults to "A1" for files being created; for files being opened, all accessed disks are searched and the CMS filemode is set to that of the first file with the appropriate filename and filetype. If, in addition, the filetype is omitted it defaults to "FILE". The case of the characters of a CMS file name is not significant.

DDNAME

The NAME parameter may also be a DDNAME. If the file name parameter starts with a % character, the remainder of the string (excluding trailing blanks) is taken as a DDNAME previously specified using the CMS FILEDEF command. If the DDNAME has not been specified using FILEDEF, NAME_ERROR will be raised. The case of the characters of a DDNAME is not significant.

The effect of calling CREATE and DELETE for a file opened using a DDNAME is as if OPEN or CLOSE (respectively) had been called.

In-store file name

The NAME parameter may also be an in-store file name. An in-store file name starts with a \ character but is otherwise unrestricted. The case of the characters of an in-store file name is significant.

An in-store file name represents a virtual file, which is held in memory rather than on disk. As a consequence, access to such a virtual file is more efficient than access to a disk based file. However, a virtual file has no independent external existence and will exist only until the termination of the Ada program which creates it, or until it is explicitly deleted.

8.1.2 MVS

The NAME parameter supplied to the Ada procedures CREATE or OPEN [14.2.1] may represent an MVS dataset name, a DDNAME or an in-store file name.

The syntax of the Ada NAME parameter for MVS is as follows:

dataset_name ::= *mvs_dataset_name* |
%ddname |
\store_file_name

MVS dataset name

The syntax of an MVS dataset name as specified in the Ada NAME parameter is as follows:

mvs_dataset_name ::= [&]*dsname*[(*member*)] |
'*dsname*[(*member*)]'

where

dsname is the MVS dataset name. If prefixed by an ampersand (&) the system assigns a temporary dataset name.

member is the MVS member, generation or area name.

An unqualified name (not enclosed in apostrophes) is first prefixed by the string (if any) given as the QUALIFIER parameter in the program PARM field when the program is run. An intervening period is added if required. If no QUALIFIER parameter has been specified no prefix is applied.

The QUALIFIER parameter may be specified as in the following example:

```
//STEP20 EXEC PGM=MONTHLY,PARM='/QUALIFIER(PAYROLL.ADA)'
```

A fully qualified name (enclosed in apostrophes) is not so prefixed. The result of the NAME function is always in the form of a fully qualified name, i.e. enclosed in single quotes.

DDNAME

The NAME parameter may also be a DDNAME. If the dataset name parameter starts with a % character, the remainder of the string (excluding trailing blanks) is taken as a DDNAME previously allocated. If the DDNAME has not been allocated, NAME_ERROR will be raised.

The effect of calling CREATE and DELETE for a file opened using a DDNAME is as if OPEN or CLOSE (respectively) had been called.

In-store file name

The NAME parameter may also be an in-store file name. An in-store file name starts with a \ character but is otherwise unrestricted. The case of the characters of an in-store file name is significant.

An in-store file name represents a virtual file, which is held in memory rather than on disk. As a consequence, access to such a virtual file is more efficient than access to a disk based file. However, a virtual file has no independent external existence and will exist only until the termination of the Ada program which creates it, or until it is explicitly deleted.

8.2 FORM Parameter

The FORM parameter comprises a set of attributes formulated according to the lexical rules of [2], separated by commas. The FORM parameter may be given as a null string except when DIRECT_IO is instantiated with an unconstrained type; in this case the record size attribute must be provided. Attributes are comma-separated; blanks may be inserted between lexical elements as desired. In the descriptions below the meanings of *natural*, *positive*, etc., are as in Ada; attribute keywords (represented in upper case) are identifiers [2.3] and as such may be specified without regard to case.

USE_ERROR is raised if the FORM parameter does not conform to these rules.

The attributes are as follows:

File sharing attribute

This attribute allows control over the sharing of one external file between several internal files within a single program. In effect it establishes rules for subsequent OPEN and CREATE calls which specify the same external file. If such rules are violated or if a different file sharing attribute is specified in a later OPEN or CREATE call, USE_ERROR will be raised. The syntax is as follows:

NOT_SHARED |
SHARED => *access_mode*

where

access_mode ::= READERS | SINGLE_WRITER | ANY

A file sharing attribute of:

NOT_SHARED

implies only one internal file may access the external file.

SHARED => READERS

imposes no restrictions on internal files of mode IN_FILE, but prevents any internal files of mode OUT_FILE or INOUT_FILE being associated with the external file.

SHARED => SINGLE_WRITER

is as SHARED => READERS, but in addition allows a single internal file of mode OUT_FILE or INOUT_FILE.

SHARED => ANY

places no restrictions on external file sharing.

If a file of the same name has previously been opened or created, the default is taken from that file's sharing attribute, otherwise the default depends on the mode of the file: for mode IN_FILE the default is SHARED => READERS, for modes INOUT_FILE and OUT_FILE the default is NOT_SHARED.

Record format attribute

This attribute controls the record format (RECFM) of an external file created in Ada. The attribute is only meaningful in the FORM parameter of a CREATE call; if used in the FORM parameter of an OPEN call, it will be ignored.

By default, files are created according to the following rules:

- for TEXT_IO, and instantiations of SEQUENTIAL_IO of unconstrained types, variable-length record files (RECFM = V) are created.
- for DIRECT_IO, and instantiations of SEQUENTIAL_IO of constrained types, fixed-length record files (RECFM = F) are created.

The syntax of the record format attribute is as follows:

RECFM => V | F

Record size attribute

This attribute controls the logical record length (LRECL) of an external file created in Ada. The attribute is meaningful only in the FORM parameter of a CREATE call, or in the FORM parameter of an OPEN call for a RECFM V file (variable-length record). In all other cases the attribute will be ignored.

In the case of RECFM F files (fixed-length record) the record size attribute specifies the record length of each record; in the case of RECFM V files (variable-length record), the record size attribute specifies the maximum record length which can be transferred.

In the case of DIRECT_IO.CREATE for unconstrained types the user is required to specify the record size attribute (otherwise USE_ERROR will be raised by the CREATE procedure).

In the case of DIRECT_IO and SEQUENTIAL_IO for constrained types the value given must not be smaller than ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT; USE_ERROR will be raised if this rule is violated.

In the case of DIRECT_IO and SEQUENTIAL_IO for unconstrained types the value given must not be smaller than ELEMENT_TYPE'DESCRIPTOR_SIZE / SYSTEM.STORAGE_UNIT plus the size of the largest record which is to be read or written. If a larger record is processed, DATA_ERROR will be raised by an attempted READ operation and USE_ERROR will be raised by an attempted WRITE operation.

In the case of TEXT_IO using a RECFM F file (fixed-length record), output lines will be padded to the requisite length with spaces. Trailing spaces can be ignored when reading a RECFM F file with TEXT_IO under the control of the truncate attribute.

The syntax of the record size attribute is as follows:

LRECL | RECORD_SIZE => *natural*

where *natural* is a size in bytes.

For input-output of constrained types the default is:

LRECL => *element_length*

where

element_length = ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT

For input-output of unconstrained types other than via DIRECT_IO, in which case the record size attribute must be provided by the user, variable size records are used (RECFM V).

Carriage control

This attribute applies to TEXT_IO only, and is intended for files destined to be sent to a printer.

For a file of mode OUT_FILE, this attribute causes the output procedures of TEXT_IO to place a carriage control character as the first character of every output record; '1' (skip to channel 1) if the record follows a page terminator, or space (skip to next line) otherwise. Subsequent characters are output as normal as the result of calls of the output subprograms of TEXT_IO.

For a file of mode IN_FILE, this attribute causes the input procedures of TEXT_IO to interpret the first character of each record as a carriage control character, as described in the previous paragraph. Carriage control characters are not explicitly returned as a result of an input subprogram, but will (for example) affect the result of END_OF_PAGE.

The user should naturally be careful to ensure the carriage control attribute of a file of mode IN_FILE has the same value as that specified when creating the file.

The syntax of the carriage control attribute is as follows:

CARRIAGE_CONTROL [=> *boolean*]

For CMS files, the default is set according to the filetype of the file: if the filetype is LISTING, the default is CARRIAGE_CONTROL => TRUE otherwise the default is CARRIAGE_CONTROL => FALSE. If the attribute alone is specified without a boolean value it is set to TRUE.

Truncate

This attribute applies to TEXT_IO files, and causes the input procedures of TEXT_IO to remove trailing blanks from records read.

The syntax of the truncate attribute is as follows:

TRUNCATE [=> *boolean*]

The default is TRUNCATE => FALSE for RECFM V files (variable-length record) and TRUNCATE => TRUE for RECFM F files (fixed-length record).

If the attribute alone is specified without a boolean value it is set to TRUE.

Append

This attribute causes writing to commence at the end of an existing file. When used in the FORM parameter of a CREATE call, a file represented by the given name will be opened if one already exists, otherwise a new file will be created and writing will commence at the begining of the file.

The syntax of the APPEND attribute is as follows:

APPEND [=> *boolean*]

The default is APPEND => FALSE. If the attribute alone is specified without a boolean value it is set to TRUE.

Eof string

This attribute applies only to files associated with the terminal opened using TEXT_IO, and controls the logical *end_of_file* string. If a line equal to the logical *end_of_file* string is typed in, END_OF_FILE will become TRUE. If an attempt is made to read from a file for which END_OF_FILE is TRUE, END_ERROR will be raised.

The syntax of the EOF_STRING attribute is as follows:

EOF_STRING => *sequence_of_characters*

The default is EOF_STRING => /*

The EOF_STRING may not contain commas, spaces or an equals sign (=).

If the END_OF_FILE function is called, a "look-ahead read" will be required. This means that (for example) a question-and-answer session at the terminal coded as follows:

```
while not END_OF_FILE loop
    PUT_LINE ("Enter value:");
    GET_LINE ( ... );
end loop;
```

will cause the prompt to appear only after the first value has been input. If the example is recoded without the explicit call to END_OF_FILE (but perhaps within a handler for END_ERROR) the behaviour will be appropriate.

8.2.1 MVS specific FORM attributes

The following additional FORM parameter attributes apply only to programs run under MVS. If used in programs run under VM/CMS they will be ignored. Under MVS, they are used to control the initial allocation of a dataset and apply only to calls of the CREATE procedure. If used in the form parameter of an OPEN call they have no effect.

Block size attribute

This attribute controls the block size of an external file. The block size must be at least as large as the record size (if specified) or must obey the same rules for specifying the record size.

The default is

BLOCK_SIZE => *record_size*

for RECFM F files and

BLOCK_SIZE => 4096

for RECFM V files.

Unit attribute

This attribute allows control over the unit on which a file is allocated. The syntax is as follows:

UNIT => *unit_name*

where *unit_name* specifies a group name, a device type or a specific unit address.

The default is the local installation specific default.

Volume attribute

This attribute allows control over the volume on which a file is allocated. The syntax is as follows:

VOLUME => *volume_name*

where *volume_name* specifies the volume serial number.

The default is the local installation specific default.

Primary attribute

This attribute allows control over the primary space allocation for a file. The syntax is as follows:

PRIMARY => *natural*

where *natural* is the number of blocks allocated to the file.

The default is the local installation specific default.

Secondary attribute

This attribute allows control over the secondary space allocation for a file. The syntax is as follows:

SECONDARY => *natural*

where *natural* is the number of additional blocks allocated to the file if more space is needed.

The default is the local installation specific default.

8.3 STANDARD_INPUT and STANDARD_OUTPUT

The Ada internal files STANDARD_INPUT and STANDARD_OUTPUT are associated with the external files %ADAIN and %ADAOUT, respectively. By default under CMS the DDNAMEs ADAIN and ADAOUT are defined to be the terminal, but the user may redefine their assignments using the FILEDEF command before running any program. Under MVS and MVS/XA, the DDNAMES must be allocated before any program is run, whether or not the corresponding Ada internal files are used.

8.4 USE_ERROR

The following conditions will cause USE_ERROR to be raised:

- Specifying a FORM parameter whose syntax does not conform to the rules given above.
- Specifying the EOF_STRING FORM parameter attribute for files other than TEXT_IO files.
- Specifying the CARRIAGE_CONTROL FORM parameter attribute for files other than TEXT_IO files.
- Specifying the BLOCK_SIZE FORM parameter attribute to have a value less than RECORD_SIZE.
- Specifying the RECORD_SIZE FORM parameter attribute to have a value of zero, or failing to specify RECORD_SIZE for instantiations of DIRECT_IO for unconstrained types.
- Specifying a RECORD_SIZE FORM parameter attribute to have a value less than that required to hold the element for instantiations of DIRECT_IO and SEQUENTIAL_IO for constrained types.
- Violating the file sharing rules stated above.
- For CMS, attempting to write a zero length record to other than the terminal.
- Errors detected whilst reading or writing (e.g. writing to a file on a read-only disk).

8.5 TEXT TERMINATORS

Line terminators [14.3] are not implemented using a character, but are implied by the end of physical record.

Page terminators [14.3] are implemented using the EBCDIC character 0C (hexadecimal) when the CARRIAGE_CONTROL FORM attribute is FALSE, and by using the first character of each record when CARRIAGE_CONTROL is TRUE.

File terminators [14.3] are not implemented using a character, but are implied by the end of physical file. Note that for terminal input a line consisting of the EOF_STRING (see 8.2) is interpreted as a file terminator. Thus, entering such a line to satisfy a read from the terminal will raise the END_ERROR exception.

The user should avoid the explicit output of the character ASCII.FF [C], as this will not cause a page break to be emitted. If the user explicitly outputs the character ASCII.LF, this is treated as a call of NEW_LINE [14.3.4].

The following characters have special meaning for VM/SP; this should be borne in mind when reading from the display terminal:

Character	Default VM/SP meaning	May be changed using
#	logical line end symbol	CP TERMINAL LINEND
"	logical escape character	CP TERMINAL ESCAPE
@	logical character delete symbol	CP TERMINAL CHARDEL

8.6 EBCDIC and ASCII

All I/O using TEXT_IO is performed using ASCII/EBCDIC translation. CHARACTER and STRING values are held internally in ASCII but represented in external files in EBCDIC. For SEQUENTIAL_IO and DIRECT_IO no translation takes place, and the external file contains a binary image of the internal representation of the Ada element (see section 8.7).

It should be noted that the EBCDIC character set is larger than the (7 bit) ASCII and that the use of EBCDIC and ASCII control characters may not produce the desired results when using TEXT_IO (the input and output of control characters is in any case not defined by the Ada language [14.3]). Furthermore, the user is advised to exercise caution in the use of BAR (|) and SHARP (#), which are part of the lexis of Ada; if their use is prevented by translation between ASCII and EBCDIC, EXCLAM (!) and COLON (:), respectively, should be used instead [2.10].

Various translation tables exist to translate between ASCII and EBCDIC. The predefined package EBCDIC is provided to allow access to the translation facilities used by TEXT_IO and SYSTEM_ENVIRONMENT (see *Character Code Translation Tables* in the *Compiler User's Guide*).

The specification of this package is given in the *Application Developer's Guide*, Section 4.1

8.7 Characteristics of Disk Files

A disk file that has already been created and is opened takes on the characteristics that are already associated with that file.

The characteristics of disk files that are created using the predefined input-output packages are set up as described below.

8.7.1 TEXT_IO

- A carriage control character is placed in column 1 if the carriage control attribute is specified in the FORM parameter.
- Data is translated between ASCII and EBCDIC so that the external file is readable using other System/370 tools.
- Under MVS and MVS/XA, TEXT_IO files are implemented as DSORG PS (QSAM) datasets.

8.7.2 SEQUENTIAL_IO

- No translation is performed between ASCII and EBCDIC; the data in the external file is a memory image of the elements written, preceded by a descriptor in the case of unconstrained types.
- Under MVS and MVS/XA, SEQUENTIAL_IO files are implemented as DSORG PS (QSAM) datasets.

8.7.3 DIRECT_IO

- No translation is performed between ASCII and EBCDIC; the data in the external file is a memory image of the elements written, preceded by a descriptor in the case of unconstrained types.
- Under CMS DIRECT_IO files may be read using SEQUENTIAL_IO (and vice-versa if a RECORD_SIZE component is specified).
- Under MVS and MVS/XA, DIRECT_IO files are implemented as DSORG DA (BDAM) datasets. The first record contains the total number of records on the first four bytes.

CHAPTER 9

Characteristics of Numeric Types

9.1 Integer Types

The ranges of values for integer types declared in package STANDARD are as follows:

```
SHORT_SHORT_INTEGER     -128 .. 127                 -- -2**7 .. 2**7 - 1

SHORT_INTEGER           -32768 .. 32767             -- -2**15 .. 2**15 - 1

INTEGER                 -2147483648 .. 2147483647   -- -2**31 .. 2**31 - 1
```

For the packages DIRECT_IO and TEXT_IO, the ranges of values for types COUNT and POSITIVE_COUNT are as follows:

```
COUNT                   0 .. 2147483647             -- 0 .. 2**31 - 1

POSITIVE_COUNT          1 .. 2147483647             -- 1 .. 2**31 - 1
```

For the package TEXT_IO, the range of values for the type FIELD is as follows:

```
FIELD                   0 .. 255                    -- 0 .. 2**8 - 1
```

9.2 Floating Point Type Attributes

SHORT_FLOAT

		Approximate value
DIGITS	6	
MANTISSA	21	
EMAX	84	
EPSILON	2.0 ** -20	9.54E-07
SMALL	2.0 ** -85	2.58E-26
LARGE	2.0 ** 84 * (1.0 - 2.0 ** -21)	1.93E+25
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 252 * (1.0 - 2.0 ** -21)	7.24E+75
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -24)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -24)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	6	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	FALSE	
SIZE	32	

FLOAT

		Approximate value
DIGITS	15	
MANTISSA	51	
EMAX	204	
EPSILON	2.0 ** -50	8.88E-16
SMALL	2.0 ** -205	1.94E-62
LARGE	2.0 ** 204 * (1.0 - 2.0 ** -51)	2.57E+61
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 252 * (1.0 - 2.0 ** -51)	7.24E+75
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -56)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -56)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	14	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	FALSE	
SIZE	64	

LONG_FLOAT

		Approximate value
DIGITS	18	
MANTISSA	61	
EMAX	244	
EPSILON	2.0 ** -60	8.67E-19
SMALL	2.0 ** -245	1.77E-74
LARGE	2.0 ** 244 * (1.0 - 2.0 ** -61)	2.83E+73
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 252 * (1.0 - 2.0 ** -61)	7.24E+75
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -112)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -112)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	28	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	FALSE	
SIZE	128	

9.3 Attributes of Type DURATION

DURATION'DELTA	2.0 ** -14
DURATION'SMALL	2.0 ** -14
DURATION'LARGE	131072.0
DURATION'FIRST	-131072.0
DURATION'LAST	131071.0

CHAPTER 10

Other Implementation-Dependent Characteristics

10.1 Characteristics of the Heap

All objects created by allocators go into the program heap. In addition, portions of the Ada Run-Time Executive's representation of task objects, including the task stacks, are allocated in the program heap.

All objects on the heap belonging to a given collection have their storage reclaimed on exit from the innermost block statement, subprogram body or task body that encloses the access type declaration associated with the collection. For access types declared at the library level, this deallocation occurs only on completion of the main program.

There is no further automatic storage reclaimation performed, i.e. in effect, all access types are deemed to be controlled [4.8]. The explicit deallocation of the object designated by an access value can be achieved by calling an appropriate instantiation of the generic procedure UNCHECKED_DEALLOCATION.

Space for the heap is initially claimed from the system on program start up and additional space may be claimed as required when the initial allocation is exhausted. The size of both the initial allocation and the size of the individual increments claimed from the system may be controlled by the Binder options SIZE and INCREMENT. Corresponding run-time options also exist.

On an extended architecture machine space allocated from the program heap may be above or below the 16 megabyte virtual storage line. The implementation defined pragma RMODE (see section 1.5) is provided to control the residence mode of objects allocated from the program heap.

10.2 Characteristics of Tasks

The default initial task stack size is 16 Kbytes, but by using the Binder option TASK the size for all task stacks in a program may be set to any size from 4 Kbytes to 16 Mbytes. A corresponding run-time option also exists.

If a task stack becomes exhausted during execution, it is automatically extended using storage claimed from the heap. The TASK option specifies the minimum size of such an extension, i.e. the task stack is extended by the size actually required or by the value of the TASK option, whichever is the larger.

Timeslicing is implemented for task scheduling. The default time slice is 1000 milliseconds, but by using the Binder option SLICE the time slice may be set to any multiple of 10 milliseconds. A corresponding run-time option also exists. It is also possible to use this option to specify no timeslicing, i.e. tasks are scheduled only at explicit synchronisation points. Timeslicing is started only upon activation of the first task in the program, so the SLICE option has no effect for sequential programs.

Normal priority rules are followed for preemption, where PRIORITY values run in the range 1 .. 10. All tasks with "undefined" priority (no pragma PRIORITY) are considered to have a priority of 0.

The minimum timeable delay is 10 milliseconds.

The maximum number of active tasks is limited only by memory usage. Tasks release their storage allocation as soon as they have terminated.

The acceptor of a rendezvous executes the accept body code in its own stack. A rendezvous with an empty accept body (e.g. for synchronisation) need not cause a context switch.

The main program waits for completion of all tasks dependent on library packages before terminating. Such tasks may select a terminate alternative only after completion of the main program.

Abnormal completion of an aborted task takes place immediately, except when the abnormal task is the caller of an entry that is engaged in a rendezvous. Any such task becomes abnormally completed as soon as the rendezvous is completed.

If an aborted task is in another MVS system process, then the abort is guaranteed to take effect by the next synchronisation point [9.10].

If a global deadlock situation arises because every task (including the main program) is waiting for another task, the program is aborted and the state of all tasks is displayed.

10.3 Definition of a Main Program

A main program must be a non-generic, parameterless, library procedure.

10.4 Ordering of Compilation Units

The Alsys IBM 370 Ada Compiler imposes no additional ordering constraints on compilations beyond those required by the language.

INDEX